



5 Chemin des Presses, 4 allée Technopolis  
06800 Cagnes sur Mer  
FRANCE

# **IQN2 LLD**

## **Software Design Specification (SDS)**

Revision A

### **Document License**

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

### **Contributors to this document**

Copyright (C) 2012 Texas Instruments Incorporated - <http://www.ti.com/>

Revision Record	
Document Title: <b>Software Design Specification</b>	
Revision	Description of Change
A	1. Initial Release – Code drop 1.0.0.0

Note: Be sure the Revision of this document matches the Approval record Revision letter. The revision letter increments only upon approval via the Quality Record System.

## TABLE OF CONTENTS

<b>1</b>	<b>SCOPE .....</b>	<b>2</b>
<b>2</b>	<b>REFERENCES .....</b>	<b>2</b>
<b>3</b>	<b>DEFINITIONS .....</b>	<b>2</b>
<b>4</b>	<b>OVERVIEW .....</b>	<b>4</b>
4.1	HARDWARE OVERVIEW .....	4
4.2	SOFTWARE OVERVIEW .....	6
4.3	KEY FEATURES .....	6
<b>5</b>	<b>DESIGN .....</b>	<b>7</b>
5.1	ARCHITECTURE .....	7
5.2	COMPLEXITY ABSTRACTION AND FLEXIBILITY .....	8
5.3	IQN2 DRIVER CONFIGURATION OBJECT .....	8
5.4	IQN2 DRIVER INITIALIZATION SEQUENCE.....	10
5.5	IQN2 DRIVER EXTERNAL INTERFACE (PUBLIC APIS) .....	10
<b>6</b>	<b>IMPLEMENTATION.....</b>	<b>12</b>
6.1	PROTOCOL.....	12
6.1.1	Definition .....	12
6.2	ANTENNA CONTAINERS (AXCS).....	12
6.2.1	Definition .....	12
6.2.2	Properties .....	12
6.2.3	First AxC and number of AxCs .....	14
6.3	RADIO STANDARDS.....	15
6.3.1	Definition .....	15
6.3.2	Properties .....	15
6.3.3	Grouping.....	16
6.3.4	LTE specifics.....	17
6.3.4.1	MBSFN.....	17
6.4	RADIO TIMERS.....	17
6.4.1	Definition .....	17
6.4.2	Properties .....	17
6.4.3	Egress/Ingress radio timers .....	18
6.4.4	LTE specifics.....	19
6.4.5	WCDMA specifics .....	20
6.5	AT2 EVENTS .....	20
6.5.1	Definition .....	20
6.5.2	Properties .....	21
6.5.3	Initialization of AT2 events .....	22
6.6	CPRI/OBSAI AIL.....	22
6.6.1	Definition .....	22
6.6.2	Properties .....	22
6.6.3	Tx wait delay.....	24
6.6.4	Cpri packing mode.....	25
6.6.5	AIL physical Timing parameters.....	25
6.6.5.1	Pi value .....	26
6.6.6	Data trace .....	26

---

6.7	DFE AID2 .....	26
<b>6.7.1</b>	<i>Definition</i> .....	26
<b>6.7.2</b>	<i>Properties</i> .....	26
6.8	DIRECTIO ENGINES (DIO2).....	28
<b>6.8.1</b>	<i>Definition</i> .....	28
<b>6.8.2</b>	<i>Ingress and Egress parameters</i> .....	28
<b>6.8.3</b>	<i>Properties</i> .....	29
<b>6.8.4</b>	<i>Duplicate feature</i> .....	30
<b>6.8.5</b>	<i>RAC use case</i> .....	30
<b>6.8.6</b>	<i>TAC use case</i> .....	30
<b>6.8.7</b>	<i>DIO reconfiguration using IQN2 Functional Layer</i> .....	31
6.9	CONTROL CHANNELS.....	31
<b>6.9.1</b>	<i>Definition</i> .....	31
<b>6.9.2</b>	<i>Properties</i> .....	31
6.10	SYNCHRONIZATION PROCESS .....	32
<b>6.10.1</b>	<i>Mechanism</i> .....	32
<b>6.10.2</b>	<i>SW support</i> .....	32
6.11	RUNTIME CONFIGURATION.....	33
<b>6.11.1</b>	<i>AIL Enable/Disable</i> .....	33
<b>6.11.2</b>	<i>AxC Enable/Disable</i> .....	33
<b>6.11.3</b>	<i>AT2 event Enable/Disable</i> .....	33
6.12	EXCEPTION COUNTERS .....	33

**LIST OF FIGURES**

Figure 1: IQN2 Hardware Block Diagram ..... 4  
Figure 2: IQN2 LLD Software Overview ..... 6  
Figure 3: IQN2 LLD Architecture..... 7  
Figure 4: IQN2 Driver data structures ..... 9  
Figure 5: IQN2 Driver Initialization Sequence ..... 10

## 1 Scope

This document describes the IQN2 Low Level Driver design. Also, the data types, data structures and application programming interfaces (APIs) provided by the IQN2 driver are explained in this document.

## 2 References

The following references are related to the feature described in this document and shall be consulted as necessary.

No	Referenced Document	Control Number	Description
1	IQN2 User Guide	SPRUxxx	KeyStone II Architecture IQN2 User Guide
2	IQN2 LLD API Documentation	Version 1.0.0.0	DOXYGEN generated API documentation located in the package under the “docs” directory in CHM format.

**Table 1. Referenced Materials**

## 3 Definitions

Acronym	Description
AID	Antenna Interface for DFE
AIF	Antenna Interface
AIL	Antenna Interface Link
API	Application Programming Interface
AT	Antenna Timer
CSL	Chip Support Library
CPRI	Common Public Radio Interface
CPU	Central Processing Unit
C&M	Control and Management
DFE	Digital Front End
DIO	Direct I/O
DMA	Direct Memory Access
DRFE	Digital Radio Front End
DSP	Digital Signal Processor
EDC	Egress DMA Controller

Acronym	Description
EE	Error/Exception Event
EFE	Egress Framing Engine
FIFO	First In First Out
FFTC	Fast Fourier Transform Coprocessor
IDC	Ingress DMA Controller
IFE	Ingress Framing Engine
IP	Intellectual Property
IQ	In-phase and Quadrature data
IQN2	In-phase and Quadrature Net v2
IQS	IQ Switch Infrastructure
ISR	Interrupt Service Routine
LLD	Low Level Driver
MCSDK	Multi-Core Software Development Kit
MMR	Memory Mapped Register
NetCP	Network Co-Processor
OBSAI	Open Base Station Architecture Initiative
OSAL	Operating System Abstraction Layer
PA	Packet Accelerator
PktDMA	Packet Direct Memory Access
PSI	Packet Streaming Interface
QMSS	Queue Manager Sub-System
RAC	Receive Accelerator Coprocessor
RAT	Radio Access Technology
SDK	Software Development Kit
SI	IQN System Interface
SOC	System On Chip
TAC	Transmit Accelerator Coprocessor
TI	Texas Instruments Inc.

**Table 2. Definitions**

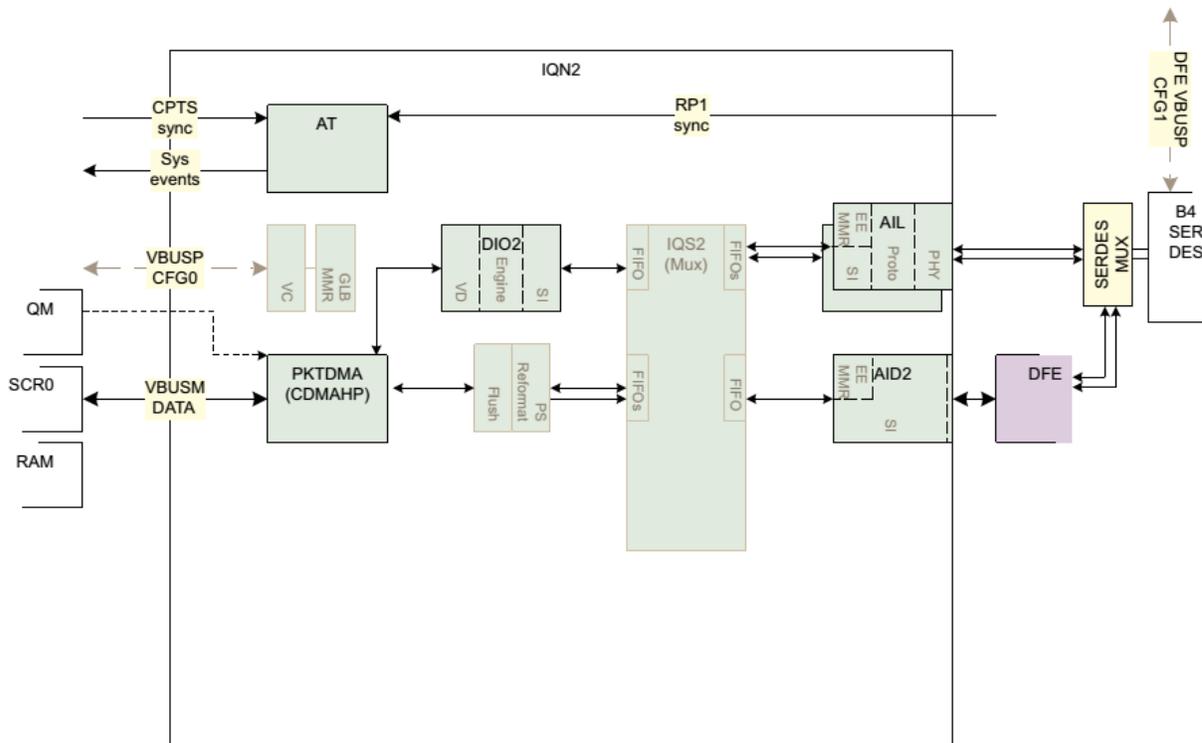
## 4 Overview

IQN2 is intended to communicate with FFTC, RAC, TAC, PA/NetCP sub-systems without the need of constant supervision or control from application software. It is envisioned that application software would initially configure the interaction between IQN2 and these sub-systems, but that in steady state, no or very minimal CPU interventions are required for this data interchange.

IQN2 LLD is designed to provide a set of low level APIs which can be used for IQN2 IP hardware setup, control and status determination for different radio standard protocols over link layer and DFE. The LLD sets several IQN2 registers to default values based on the configuration provided by the application software.

IQN2 LLD also provides IQN2-FL (Functional Layer) APIs which allow setting up and configuration of individual IQN2 sub-module registers.

### 4.1 Hardware Overview



**Figure 1: IQN2 Hardware Block Diagram**

The Figure 1: IQN2 Hardware Block Diagram above shows the hardware overview of the IQN2 peripheral. The IQN2 consists of:

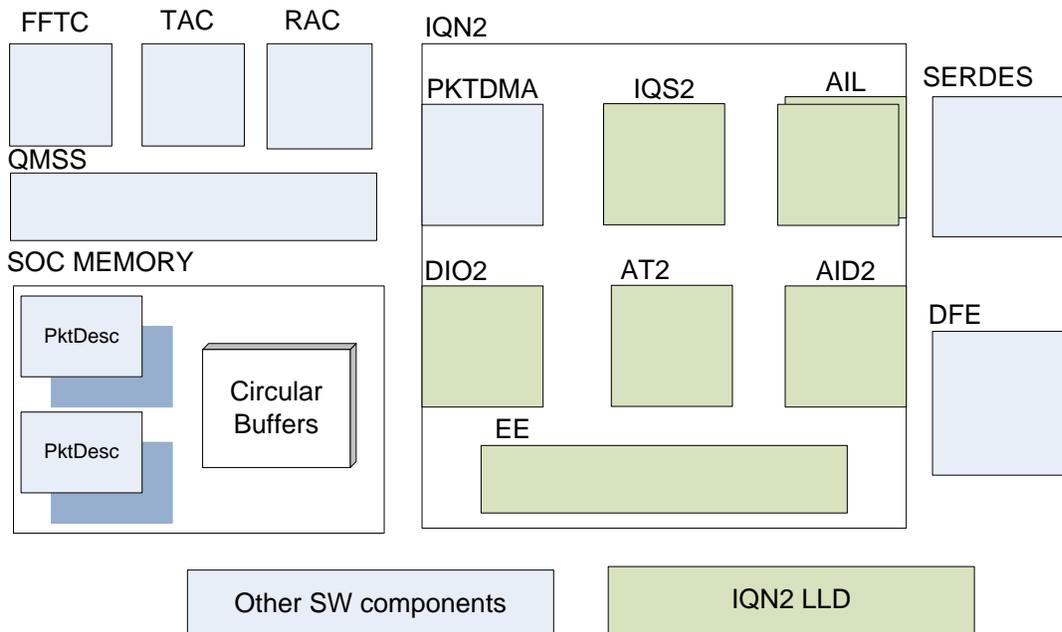
- A VBUSM DATA connection (128 bit Master) for PktDMA

- A VBUSP CFG connection (32 bit Slave) for IQN2 register access
- A PktDMA module with:
  - 48 ingress/egress channels
  - 48 quadword buffer per channel
  - 48 QPEND signals, 32 for AxC channels, 16 for control channels
  - The allocation between control and AxC is soft/flexible
- A DIO2 module with:
  - 16 ingress/egress channels
  - 3 DIO engines within DIO2
  - Up to 3-way Ingress multi-cast
  - Each engine independently programmed DMA profile
  - Each engine can have different burst length and SOC addressing scheme
- A AID2 module to interface the new Digital Front End (DFE) module with:
  - 32 ingress/egress AxC channels
  - 16 ingress/egress Control channels
- Two AIL instances to handle CPRI and OBSAI protocol with
  - OBSAI
    - Up to 32 ingress/egress AxC/Ctrl mixture channels
    - link rates {2x, 4x, 8x}
  - CPRI
    - Up to 32 ingress/egress AxC channels
    - Up to 4 ingress/egress control channels
    - Link rates {2x, 4x, 5x, 8x, 10x, 16x}
  - Link-to-link PHY level forwarding (defined in OBSAI/CPRI)
  - PHY reset isolation
- An AT2 module with:
  - 24 System Events
  - 1 BCN counter
  - 8 complex RADT (radio Timers)
  - Timing Sync Sources {RP1, generic input pins, CPTS(10gE), SW}
- Power Islands for AIL modules

- CPTS with clock synchronization circuit
- A PSR/Flush Controller with peer requested flush support
- A EE module to aggregate Alarm and Exception information and generate SoC level interrupts

The muxing of SERDES lanes between AIL (CPRI/OBSAI) and DFE (JESD) is performed at SoC level.

## 4.2 Software Overview



**Figure 2: IQN2 LLD Software Overview**

The Figure 2: IQN2 LLD Software Overview depicts the various components involved in the transfer of IQ and control data when the IQN2 driver runs on the SoC. The green boxes are the IQN2 modules that the LLD configures for the different mode of operations (LTE/WCDMA, DFE/CPRI/OBSAI). The IQN2 configuration can be run from ARM or DSP. The IQN2LLD is single instance.

The other SW components in light blue are either CSL (SERDES), LLDs (CPPI, DFE, FFTC, QMSS), or FLs (RAC, TAC).

Even though IQN2 HW includes a PktDMA instance, the PktDMA configuration and the associated Rx/Tx descriptor queues and free descriptor queues is not owned by IQN2 LLD. This allows, for instance, to have multiple SW entities running on ARM or DSP to push or pop packets to/from IQN2, each SW entity being in charge of configuring its separate Rx/Tx PktDMA channels.

## 4.3 Key Features

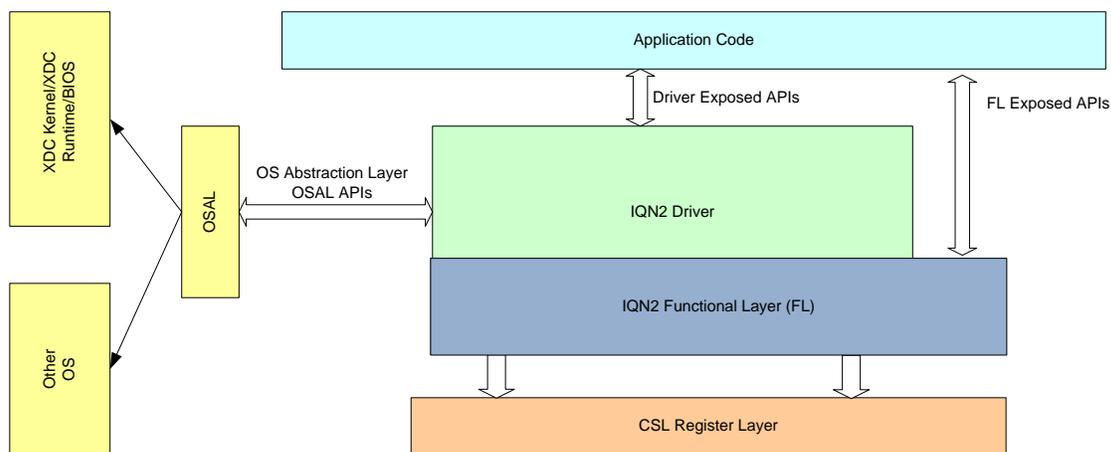
Following are the key features of IQN2 LLD software:

- IQN2 Functional Layer on top of CSLR (register layer) for IQN2 sub-modules (AIL, Top, AID2, IQS2, DIO2, AT2, and all EEs)
- Single instance LLD driver for DSP and ARM (user mode) with several mode of operations, including:
  - LTE, WCDMA, Dual mode, and Generic Packet mode
  - CPRI over AIL, including CPRI control words
  - OBSAI over AIL
  - DFE over AID2 (AxC/CTL)

## 5 Design

### 5.1 Architecture

This section explains the overall architecture of IQN2 LLD:



**Figure 3: IQN2 LLD Architecture**

The [Figure 3: IQN2 LLD Architecture](#) illustrates the following key components:

#### 1.) IQN2 Driver

This is the core IQN2 driver. The driver exposes a set of APIs which are used by the application layer to configure, monitor, reconfigure and reset the IQN2 peripheral module. The driver also exposes a set of OS abstraction APIs which are used to ensure that the driver is OS independent and portable. The IQN2 driver uses its IQN2 functional layer for all IQN2 MMR access.

#### 2.) Application Code

This is the user of the driver and its interface with the driver is through the APIs set. Application users use the driver APIs to configure, monitor, reconfigure and reset the IQN2 module.

#### 3.) Operating System Abstraction Layer (OSAL)

The IQN2 LLD is OS independent and exposes all the operating system callouts via this OSAL layer.

#### 4.) IQN2 Functional Layer

The IQN2 Functional Layer is used to setup IQN2 HW via a top-level FL configuration structure which exposes all the IQN2 low-level fields. It also offers a set of specific commands and queries for each IQN2 sub-modules (AIL, Top, AID2, IQS2, DIO2, AT2, and all EEs).

#### 5.) CSL Register Layer

The CSL register layer is the IP block memory mapped registers which are generated by the IP owner. CSLR names are matching the IP HW specification. The IQN2 FL accesses the MMR registers via CSLR.

## 5.2 Complexity abstraction and flexibility

This IQN2 LLD aims at generalizing the configuration of IQN2 for different high-level scenarios. Only a single instance of the driver can be run from only one of the KeyStone-II SOC ARM or DSP cores. So the LLD should be seen as an abstraction of the IQN2 configuration complexity. That means that within a short amount of configuration parameters / API calls, the application code can configure IQN2 for a specific high level scenario. Enough flexibility is put in the LLD at this point to achieve that goal. The LLD implementation actually won't prevent the application code from overriding the "pre-defined" parameters. APIs have been split in such a way that first, the FL high level setup structure (Iqn2Fl\_Setup) of the IQN2 driver instance gets populated (IQN2\_initHw):

```
// initialization function for the IQN2 H/W FL structure
IQN2_initHw(&iqn2Obj);
```

And then the Iqn2Fl\_Setup configuration is applied to IQN2 HW registers in a separate API call.

```
IQN2_startHw(&iqn2Obj);
```

That means iqn2Obj.hIqn2Setup can still be altered in between with the "end user" very specific needs by using a set of LLD APIs.

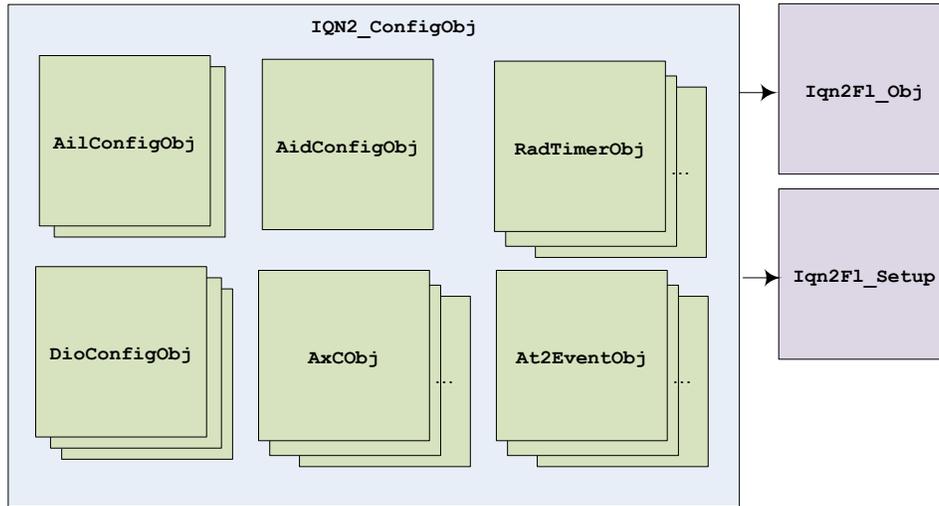
## 5.3 IQN2 Driver configuration object

The IQN2 LLD defines a structure called IQN2\_ConfigObj. It exposes high level parameters to allow users to configure IQN2 with a general understanding of how IQN2 HW works.

IQN2\_ConfigObj fields are, for instance, the type of antenna traffic, the number of links, the protocol, the data width, the DMA mode etc...

IQN2 LLD is then in charge of translating these parameters into IQN2 FL setup parameters, and then call the appropriate FL APIs to setup IQN2 registers.

The Driver configuration object, besides its own properties, contains configuration objects for IQN2 modules such as AIL, DIO2, or AID. It also contains objects with each Antenna Container (AxC) and each Radio Timer (RADT). IQN2\_ConfigObj has also pointer references to an Iqn2F1 object instance and the Iqn2F1 setup structure which is used to configure IQN2 HW from IQN2\_startHw() API call.



**Figure 4: IQN2 Driver data structures**

### 5.4 IQN2 Driver initialization sequence

The IQN2 driver provides a sequence which initializes the IQN2 IP block.

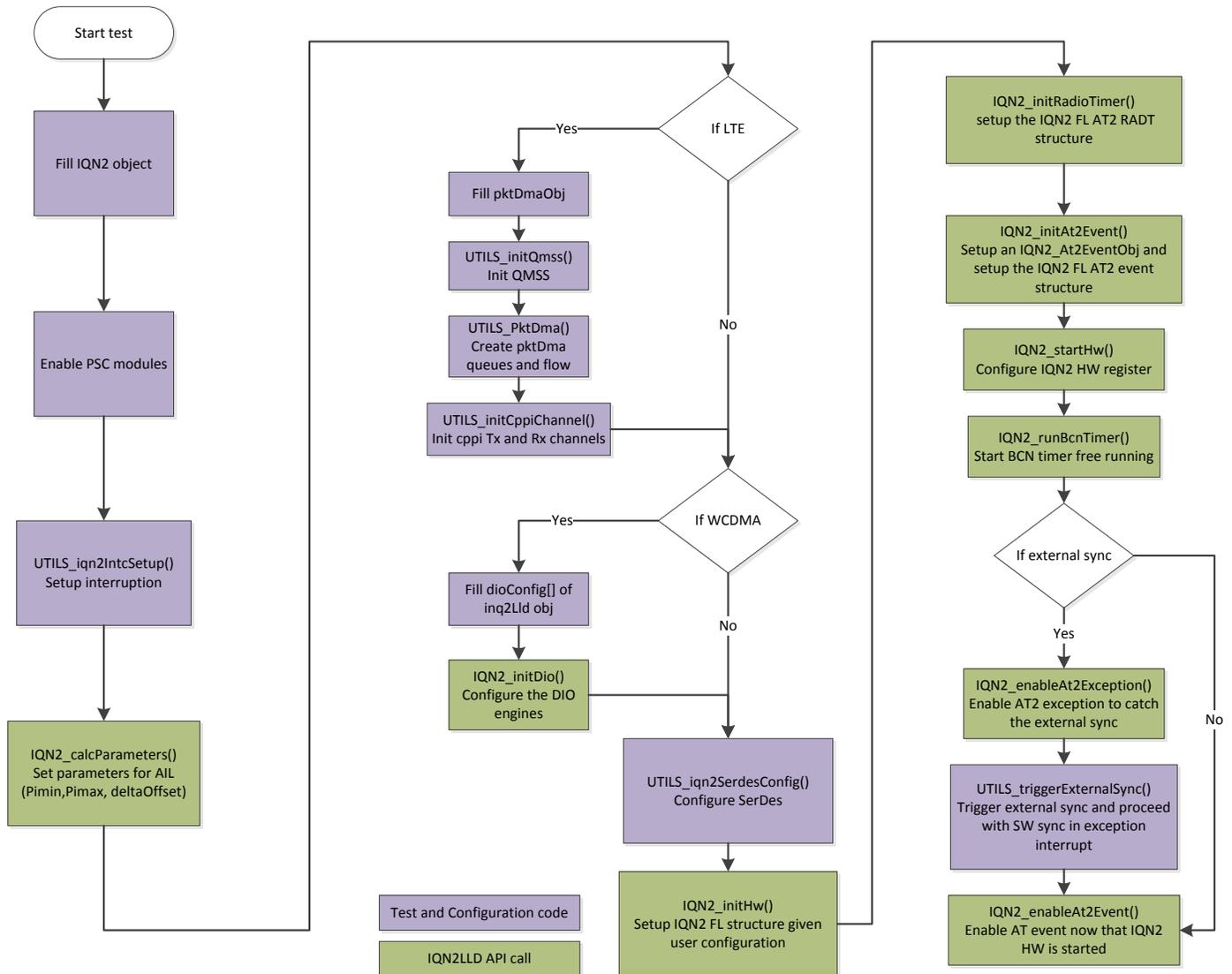


Figure 5: IQN2 Driver Initialization Sequence

The Figure 5: IQN2 Driver Initialization Sequence depicts the typical control flow during the initialization of the IQN2.

### 5.5 IQN2 Driver External Interface (Public APIs)

The following table outlines the basic interfaces provided by IQN2 LLD. Please note that this list is not exhaustive. For complete list of APIs, there is doxygen documentation associated with each release of IQN2LLD.

<b>IQN2 Functional Layer</b>	<b>Description</b>
Iqn2Fl_init	Peripheral specific initialization function
Iqn2Fl_open	Opens an instance of iqn2 fl
Iqn2Fl_hwSetup	Initializes the device registers
Iqn2Fl_hwControl	Controls IQN2 operation based on the control command
Iqn2Fl_getHwStatus	Gets IQN2 status values based on the status query command
Iqn2Fl_close	Closing an instance of iqn2 fl

<b>IQN2 Driver</b>	<b>Description</b>
IQN2_calcParameters	Calculates IQN2 timing and delay parameters according to the pre-set parameters in IQN2_ConfigObj.
IQN2_initDio	Sets up the dio configuration structure given the application parameters.
IQN2_initHw	Sets up the IQN2 FL setup structures given the IQN2_ConfigObj instance
IQN2_startHw	Configures IQN2 HW registers given user IQN2 configuration. HW is then started, waiting for the selected synchronization
IQN2_runBcnTimer	Starts the BCN Timer free running. SW writes are not precise so it is expected in real case system, the application SW will correct the timer value with an offset. IQN2_resyncProcedure is used up on a resync external event.
IQN2_initRadioTimer	Sets up the IQN2 FL setup Radio Timer structures given the IQN2_ConfigObj instance, and the egress/ingress radio standards configured during IQN2_initHw().
IQN2_initAt2Event	Sets up the IQN2 FL setup AT2 event structures given the IQN2_At2Eventobj instance. This instance is also registered in IQN2_ConfigObj instance.
IQN2_enableAt2Event	Enables a given AT2 event, assuming this event was already initialized
IQN2_disableAt2Event	Disables a given AT2 event
IQN2_getIngressRadioStandardId	Returns the ingress radio standard ID associated with the AxC
IQN2_getEgressRadioStandardId	Returns the egress radio standard ID associated with the AxC. Used for dynamic control of MBSFN in LTE
IQN2_getIngressRadioTimerId	Returns the ingress radio timer ID associated with the AxC number. Ingress radio timer id is equal to Ingress radio standard id + max number of populated radio standards. Application may need to have separate radio timers for downlink and uplink. Following call to IQN2_initHw(), with this function, the application can get a radio timer ID and pass user-

	specified parameters for radio timers in IQN2_ConfigObj
<code>IQN2_getEgressRadioTimerId</code>	Returns the egress radio timer ID associated with the AxC number. Egress radio timer id is equal to Egress radio standard id. Application may need to have separate radio timers for downlink and uplink. Following call to <code>IQN2_initHw()</code> , with this function, the application can get a radio timer ID and pass user-specified parameters for radio timers in <code>IQN2_ConfigObj</code> .
<code>IQN2_enableException</code>	Configures IQN2 HW registers to enable errors at alarms at IQN2 level
<code>IQN2_resetException</code>	Resets IQN2LLD exception counters
<code>IQN2_captureException</code>	Captures the exception counts into a supplied destination storage
<code>IQN2_resetIqn2</code>	Used to reset IQN2 to its default state

## 6 Implementation

### 6.1 Protocol

#### 6.1.1 Definition

This is a high level parameter that defines whether the IQN2 driver is used for CPRI, OBSAI, or DFE modes. This parameter is used to select AIL0/1 versus AID2 module usage, and to compute the timing and framing values for the IQN2 modules. While it is envisioned to use AIL and AID2 simultaneously, the current scope of IQN2 driver does not comprehend that scenario.

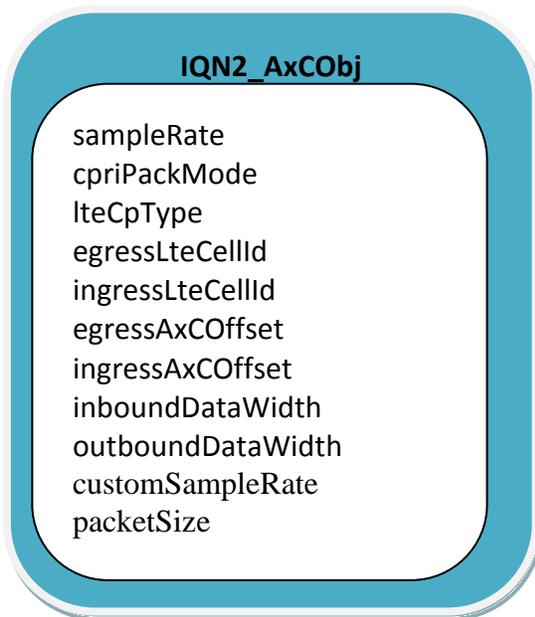
### 6.2 Antenna Containers (AxCs)

#### 6.2.1 Definition

IQN2 was designed to support flexible combinations of antenna containers (AxCs). IQN2 can support up to eight radio standards, which can result on eight different modes of operation. Each AxC can be mapped to one of these. A radio standard groups AxCs with similar properties. IQN2 LLD is allocating groups of AxCs per radio standard on its own and user application can query the radio standard id for a given AxC.

The IQN2 LLD instantiates `IQN2_AxCObj` structure objects, which holds information, for instance, about is egress or ingress data width and offset.

#### 6.2.2 Properties



sampleRate: this property is used to create a group or assigned the AxC to a specific radio standard. Each AxC of the same group holds the same sample rate. IQN2\_SRATE\_CUSTOM can be used in conjunction with customSampleRate to specify a sampling rate not part of sampleRate enumeration.

cpriPackMode: used for CPRI only. Depending on the sample rate, the antenna containers are interleaved in different order to handle the CPRI frame format.

lteCpType: LTE cyclic prefix description. The application can use normal cyclic prefix or extended cyclic prefix for each AxC. IQN2 LLD creates a different group for extended cyclic prefix if already one group is defined with the same sampling rate but normal cyclic prefix. IQN2\_LTE\_CPTYPE\_NONE can be used to implement a non-LTE framing over a 10ms radio frame. In that case, the packet size is the same over a 10ms radio frame, and packetSize needs to be specified for this AxC.

egressLteCellId: used for LTE for group creation to facilitate MBSFN operation. An egress group is required to be created for a unique combination of sampling rate and cpType. On top of this, for different Lte cell ids, different groups will be created even if their sampling rates and cotypes are identical.

ingressLteCellId: used for LTE for group creation to facilitate multi strobe with the same rate. An ingress group is required to be created for a unique combination of sampling rate and cpType. On top of this, for different Lte cell ids, different groups will be created even if their sampling rates and cotypes are identical.

egressAxCOffset: IQN2 supports per-channel AxC offsets with sample granularity except AIL in OBSAI mode where IQN2 supports per-channel AxC offsets with clock granularity. AIL allows every AxC within a group to be offset relative to the Radio Standard Offset.

ingressAxCOffset: IQN2 supports per-channel AxC offsets with sample granularity except AIL in OBSAI mode where IQN2 supports per-channel AxC offsets with clock granularity. AIL allows every AxC within a group to be offset relative to the Radio Standard Offset.

inboundDataWidth: used to specify DL/UL data formats for IQN2

outboundDataWidth: used to specify DL/UL data formats for IQN2

customSampleRate: this field can be used to specify a custom sampling rate if  $\text{sampleRate}=\text{IQN2\_SRATE\_CUSTOM}$ . It is expressed in KHz and the DFE PLL clock (2457600 or 3686400) needs to be a multiple of the custom sampling rate.

packetSize: this field can be used to pass a custom packet size if  $\text{sampleRate}=\text{IQN2\_SRATE\_CUSTOM}$ ,  $\text{customSampleRate}$  is set, and  $\text{teCpType}=\text{IQN2\_CPTYPE\_NONE}$ . It is expressed in number of samples (IQ). The associated custom sampling rate over 10ms needs to be an integer multiple of the custom packet size. Example:  $92160 * 10 / 4096 = 225$  for a sampling rate of 92.16 and a packet size of 4096 IQ samples.

### 6.2.3 First AxC and number of AxCs

There are restrictions introduced by the IQN2 LLD on the AxC mapping. IQN2 LLD support multiple modes of operation at the same time and on the same module (AIL/AID). The supported list of modes includes WCDMA and LTE (LTE 5, LTE 10, LTE 15 and LTE 20).

For each mode, the user application provides the numbers of AxCs to configure. The IQN2 LLD has restrictions on the order each AxC is allocated.

`IQN2_AilConfigObj` and `IQN2_AidConfigObj` structure objects contain, for instance, in AID2 case, a `firstLteAxC` and `numLteEgressAxC/numLteIngressAxC` parameters.

The user application needs to make sure that AxC numbers for different modes do not collide.

For instance, if Wcdma AxCs are configured first, that means:

$$\text{firstLteAxC} = \text{firstWcdmaAxC} + \max(\text{numWcdmaEgressAxC}, \text{numWcdmaIngressAxC})$$

For scenarios with a mix, for instance, of lte5, lte10, and lte20, the user application needs to try and group same types of AxCs in contiguous entries of `AxCconfig[]` array. That will minimize unnecessary creation of new radio standard group ids, knowing IQN2 allows up to 8 simultaneous radio standards. The LLD internally browses `AxCconfig[]` array and creates new groups of contiguous AxCs objects with the same properties.

So let's illustrate with an example:

#### **User application:**

```
// Suppose the following AIL0 configuration (numAil=0)
// numWcdmaAxC = 16, firstWcdmaAxC = 0, numLte20AxC = 2, firstLteAxC = 16
hIqn2->ailConfig[numAil].numWcdmaPeAxC = numWcdmaAxC;
hIqn2->ailConfig[numAil].numWcdmaPdAxC = numWcdmaAxC;
hIqn2->ailConfig[numAil].numLtePeAxC = numLte20AxC;
hIqn2->ailConfig[numAil].numLtePdAxC = numLte20AxC;
hIqn2->ailConfig[numAil].firstWcdmaAxC = 0;
hIqn2->ailConfig[numAil].firstLteAxC = hIqn2->ailConfig[numAil].firstWcdmaAxC + numWcdmaAxC;
// Corresponding AxC array configuration, note AxC with same properties are contiguous to each
other
AxCOffset = hIqn2->ailConfig[numAil].firstWcdmaAxC;
for(i=0; i< numWcdmaAxC; i++)
{
    hIqn2->AxCconfig[AxCOffset].sampleRate = IQN2_SRATE_3P84MHZ;
    hIqn2->AxCconfig[AxCOffset].inboundDataWidth = IQN2FL_DATA_WIDTH_8_BIT;
    hIqn2->AxCconfig[AxCOffset].outboundDataWidth = IQN2FL_DATA_WIDTH_8_BIT;
    AxCOffset++;
}
AxCOffset = hIqn2->ailConfig[numAil].firstLteAxC;
for(i=0; i < numLte20AxC; i++)
{
    hIqn2->AxCconfig[AxCOffset].sampleRate = IQN2_SRATE_30P72MHZ;
    hIqn2->AxCconfig[AxCOffset].cpriPackMode = IQN2_LTE_CPRI_8b8;
    hIqn2->AxCconfig[AxCOffset].inboundDataWidth = IQN2FL_DATA_WIDTH_15_BIT;
    hIqn2->AxCconfig[AxCOffset].outboundDataWidth = IQN2FL_DATA_WIDTH_15_BIT;
    AxCOffset++;
}
```

### **IQN2LLD allocation:**

```
Egress radio standard 0: 16 Wcdma AxC, index 0 to 15
Ingress radio standard 0: 16 Wcdma AxC, index 0 to 15
Egress radio standard 1: 2 LTE20 AxC, index 16 to 17
Ingress radio standard 1: 2 LTE20 AxC, index 16 to 17}
```

## **6.3 Radio standards**

### **6.3.1 Definition**

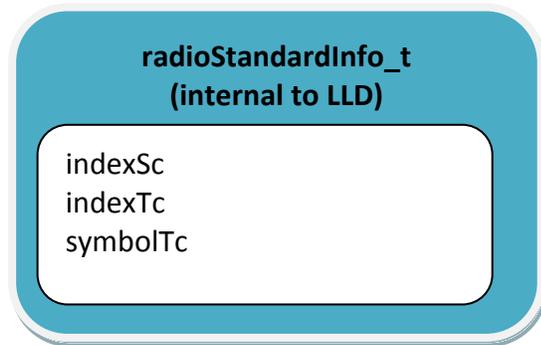
IQN2 supports the grouping of similar antenna containers (AxCs) in radio standards. Because simultaneous modes of operation (Wcdma, Lte) have different sampling rates, for instance, IQN2 needs to be able to manage several framing configurations at once. This is what the radio standards are made for. A radio standard is defined by four parameters: {INDEX\_START, INDEX\_CNT, SYMBOL\_TC and SAMPLE\_TC}. The radio standards share the same common sample Terminal Count (TC) LUT (256 entries), which the IQN2 LLD allocates based on each radio standard needs. This way, each radio standard has access to its own framing configuration values.

The allocation of radio standards is done within the LLD and is based on the description of each AxC in the AxCconfig[] array. The user application can make use of LLD APIs to get the information about the egress or ingress group id for a given AxC.

### **6.3.2 Properties**

Each Radio standards are defined by four parameters that configure their sample framing. The LLD stores these parameters in **internal** variables:

- egrRadStdInfo[IQN2\_MAX\_NUM\_RADIO\_STANDARD]
- ingrRadStdInfo[IQN2\_MAX\_NUM\_RADIO\_STANDARD].

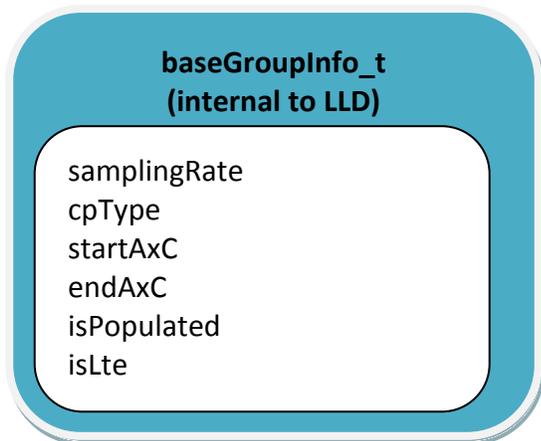


The `radioStandardInfo` structure holds the information relative to the radio framing. In IQN2 hardware implementation, all the radio standards share a common Sample Terminal Count Look Up Table of 256 entries. In order to properly allocate a radio standard, IQN2 need the first index of that LUT (`indexSc`), the number of entries (`indexTc`), and the number of symbol in one frame (`symbolTc`). The IQN2 LLD uses different entries for all the radio standards that it creates.

### 6.3.3 Grouping

The IQN2 LLD internally groups AxCs for a given radio standard. It automatically fills the groups based on the AxC parameters provided by the user application. The LLD stores each group parameters in private structure arrays:

- `egrGroupInfo[IQN2_MAX_NUM_RADIO_STANDARD]`
- `ingrGroupInfo[IQN2_MAX_NUM_RADIO_STANDARD]`



Note that the egress group info also has a “`cellId`” property, used to create different groups in Lte case for proper handling of MBSFN. The ingress group has one too for multi strobe case.

The `groupInfo` is used by the LLD to define the characteristics of each group. It holds the number of AxCs mapped on the group from the first to the last (every AxCs in a group must be contiguous). A group is created with four parameters: sampling rate, `cpType`, `cellId` and `isLte`. The LLD checks every AxCs to configure for IQN2, and it creates a group for every different combination of these three parameters. AxC with the same parameters will be mapped on the same group.

### 6.3.4 LTE specifics

#### 6.3.4.1 MBSFN

For groups of LTE AxCs with normal cyclic prefix, IQN2 LLD defines a “MBSFN factor” of 2 to use twice the number of entries in the common Sample Terminal Count Look Up Table. This allows SW to carefully modify the framing parameters. These need to be modified at runtime on transition between Normal and MBSFN Sub-Frames. To support this mechanism (described in this presentation, [http://e2e.ti.com/cfs-file.ashx/\\_key/communityserver-discussions-components-files/439/2772.MBSFN-support-on-AIF2.ppt](http://e2e.ti.com/cfs-file.ashx/_key/communityserver-discussions-components-files/439/2772.MBSFN-support-on-AIF2.ppt)), IQN2 LLD implements the following API:

```
Iqn2Fl_updateEgressAilRadioStandardTc (hIqn2, radioStdId, symbolSize, symbolIndex);
```

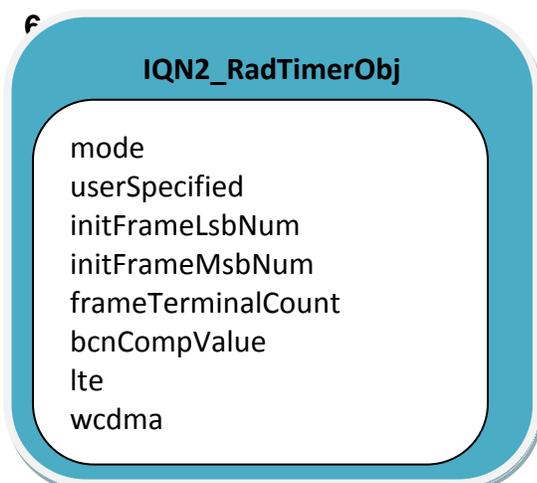
### 6.4 Radio timers

#### 6.4.1 Definition

Radio Timers (8 RADT instances in IQN2) have the ability to count radio symbols and frames and to generate AT2 events. Each radio timer is used for a given radio access type and direction, LTE or WCDMA, Downlink or Uplink. There is no hard mapping between a radio timer and a radio standard, but the LLD will do a one to one mapping between egress radio standards and radio timers. The LLD, for each mode, has default parameters as applied in IQN2\_initRadioTimer() as described in the below table. The LLD offers to override these parameters and the radio timers can be configured with user-defined parameters. In the specific case of LTE, the number of LTE symbols per symbol strobe is configurable as well as the number of LTE symbols per frame strobe. That allows an LTE application to work at 1-symbol pace, or 2-symbol pace, etc ...

In the IQN2 hardware, the different symbols of a frame are described in a common 256 entry LUT, which give a clock count length per symbol. The LLD fills this LUT based on each radio timer parameters

Radio timers are slaves to BCN timer (equivalent of PHY timer in AIF2) and are synchronized to it. BCN is expected to be started in a free-running mode by IQN2\_runBcnTimer(). For enabled RADTs, they will start running once the compare value equals the BCN timer value.



mode: IQN2 radio timer application mode, egress, ingress or unused. The LLD sets the RADT mode to egress based on the number of egress radio standards allocated during call to IQN2\_initHw()

userSpecified: used to indicate that the user application wants to override the default parameters.

initFrameLsbNum: initial frame low 32-bit number value for this radio timer.

initFrameMsbNum: initial frame high 32-bit number value for this radio timer.

frameTerminalCount: RAD timer frame terminal count (value at which frame count wraps)

bcnCompValue: BCN timer compare value used to start this RAD timer

lte: object for the userSpecified and lte specific parameters used to configure this RAD timer.

wcdma: object for the userSpecified and wcdma specific parameters used to configure this RAD timer.

List of default RAD timer parameters applied during IQN2\_initRadioTimer() when userSpecified is reset.

RADT parameters	Default values
initFrameLsbNum	1
initFrameMsbNum	0
frameTerminalCount	4096
bcnCompValue	0
Frame strobe	Every 10ms, so: LTE: 140 symbols (default is normal CP) WCDMA: 15 slots
Symbol strobe	Every symbol, so: LTE: every symbol with normal CP WCDMA: every slot (2560 chips)

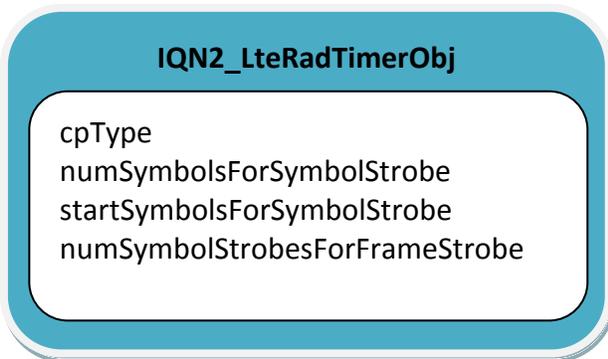
### 6.4.3 Egress/Ingress radio timers

It is expected that DL applications and UL applications make use of a different radio timer. For this matter, it is important to understand how the LLD allocated radio timers for the current number of identified simultaneous radio standards:

For DL applications, the Egress radio timer id is equal to Egress radio standard id.  
For UL applications, the Ingress radio timer id is equal to Ingress radio standard id + max number of simultaneous radio standards.

For the user application to find out about the allocated radio timer id, the LLD features 2 APIs, which take the egress or ingress AxC number as parameter and return the radio timer id: IQN2\_getEgressRadioTimerId(AxC), and IQN2\_getIngressRadioTimerId(AxC)

#### 6.4.4 LTE specifics



cpType: Lte cyclic prefix type, normal or extended, used to generate right number and frequency of the symbol strobes to the Lte applications.

numSymbolsForSymbolStrobe: This determines the terminal count array values and number of entries (lutindex). The valid range for this is 1 through 7 or 6 depending on CP type.

startSymbolsForSymbolStrobe: This parameter is unused at the moment.

numSymbolStrobesForFrameStrobe: This determines the symbol terminal count, so number of symbols per frame. So IQN2 hardware will loop thru the number of entries assigned in the common 256 entry LUT for this RAD timer and issue a frame strobe once the symbol terminal count is met.

With these parameters, the LTE applications can be paced at a 1-symbol or multi-symbol rate, or lte slot rate as well. The LLD test examples use the default parameters.

So let's illustrate with an LTE application example. In this example, one egress and one ingress RAD timers are configured. The symbol strobe is paced at 2-lte symbol rate for a given cyclic prefix type, and the frame strobe is issued at sub-frame rate. The ingress RAD timer is configured to be started taking into account the overall uplink delay:

#### User application:

```
egressRadId = IQN2_getEgressRadioTimerId(iqn2LldObj->aidConfig.firstLteAxC);
ingressRadId = IQN2_getIngressRadioTimerId(iqn2LldObj->aidConfig.firstLteAxC);

numSymbolsPerSubFrame = ((cfg->cpType == IQN2_LTE_CPTYPE_NORMAL) ? 14 : 12 );

// EGRESS
iqn2LldObj->radTimerConfig[egrRadId].mode = IQN2_RADT_EGR_MODE;
iqn2LldObj->radTimerConfig[egrRadId].userSpecified = 1;
```

```

iqn2LldObj->radTimerConfig[egrRadtId].frameTerminalCount = 10;
iqn2LldObj->radTimerConfig[egrRadtId].initFrameLsbNum = 0;
iqn2LldObj->radTimerConfig[egrRadtId].initFrameMsbNum = 0;
iqn2LldObj->radTimerConfig[egrRadtId].lte.cpType = cfg->cpType;
iqn2LldObj->radTimerConfig[egrRadtId].lte.numSymbolsForSymbolStrobe = 2;
iqn2LldObj->radTimerConfig[egrRadtId].lte.numSymbolStrobesForFrameStrobe = numSymbolsPerSubFrame/2;

// Assign the clock initial value to get an expected UL RADT delay (dependent on UL frame delay, UL AxC offset,
and internal IQN2 delays
ClockNum = radttotalclockcount - 1 - ulRadtDelay;

// INGRESS
iqn2LldObj->radTimerConfig[ingrRadtId].mode = IQN2_RADT_ING_MODE;
iqn2LldObj->radTimerConfig[ingrRadtId].userSpecified = 1;
iqn2LldObj->radTimerConfig[ingrRadtId].frameTerminalCount = 10;
iqn2LldObj->radTimerConfig[ingrRadtId].initFrameLsbNum = 0;
iqn2LldObj->radTimerConfig[ingrRadtId].initFrameMsbNum = 0;
iqn2LldObj->radTimerConfig[ingrRadtId].lte.cpType = cfg->cpType;
iqn2LldObj->radTimerConfig[ingrRadtId].lte.numSymbolsForSymbolStrobe = 2;
iqn2LldObj->radTimerConfig[ingrRadtId].lte.numSymbolStrobesForFrameStrobe = numSymbolsPerSubFrame/2;
iqn2LldObj->radTimerConfig[ingrRadtId].bcnCompValue = ClockNum;

```

## 6.4.5 WCDMA specifics

### IQN2\_WcdmaRadTimerObj

numSlotStrobesForFrameStrobe

**numSlotStrobesForFrameStrobe:** Determines the slot terminal count, so number of slots per frame - can be used to adjust the frame time to WCDMA TTI bigger than 10ms.

So let's illustrate with a WCDMA application example. The following configuration can be used to generate a 40ms event:

### User application:

```

radtId = IQN2_getIngressRadioTimerId(iqn2LldObj->aidConfig.firstWcdmaAxC);

iqn2LldObj->radTimerConfig[radtId].mode = IQN2_RADT_ING_MODE;
iqn2LldObj->radTimerConfig[radtId].userSpecified = 1;
iqn2LldObj->radTimerConfig[radtId].frameTerminalCount = 4096;
iqn2LldObj->radTimerConfig[radtId].initFrameLsbNum = 1;
iqn2LldObj->radTimerConfig[radtId].initFrameMsbNum = 0;
iqn2LldObj->radTimerConfig[radtId].wcdma.numSlotStrobesForFrameStrobe = 4*15;
IQN2_initRadioTimer (&iqn2LldObj);

// use At Event 0, init here and enable after IQN2_startHw()
memset(&at2Evt0, 0, sizeof(at2Evt0));
at2Evt0.EventSelect = IQN2_AT2_EVENT_0; // Evt 0
at2Evt0.EventOffset = 0; // Offset from frame boundary in ns
at2Evt0.EvtStrobeSel = IQN2FL_RADT1_FRAME; // Radio timer 1, frame strobe
at2Evt0.EventModulo = -1; // max modulo or wcdma frame
at2Evt0.EventMaskLsb = 0xFFFFFFFF8; // skip 3 first events
at2Evt0.EventMaskMsb = 0xFFFFFFFF;
IQN2_initAt2Event (&iqn2LldObj, &at2Evt0);

```

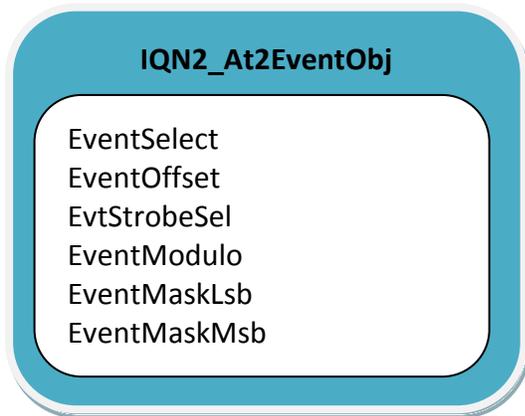
## 6.5 AT2 events

### 6.5.1 Definition

AT2 events are used by SW applications, EDMA channels, and hardware accelerators as system events for CPU interrupts, transfer triggers, or hardware processing ticks. There are a total of 24 events. AT2 event0 is connected to the EXTFRAMEEVENT external pin and can be used to

synchronize with external test equipment. AT2 event8 is dedicated for the RAC hardware accelerator and AT2 event 9 for TAC. These events are referenced to the radio timers (RADT timers), and user applications need to specify the radio timer id, and the type of strobe (symbol/frame), along with offset and modulo values. The offset and modulo values are expressed in byte clock unit, but user applications can first express those in nanosecs and get a conversion to byte clocks using `IQN2_initNanoSecsToByteClocks()` API. If 100% accuracy is required for the modulo or offset values, then the user application needs to work with byte clocks directly while configuring each of the AT2 events.

### 6.5.2 Properties



**EventSelect:** this LLD object instance contains an array of 24 AT2 event objects. There is no hard mapping between the index in the array and the AT2 event itself. That's why this parameter can be any of the 24 possible events.

**EventOffset:** `Event_offset` in Byte Clocks after the start of frame/sym strobcs. From IQN2 hardware stand point there is a `sys_event` generator that counts out `OFFSET` clocks then fires once. For convenience, the LLD provides a Nanosecs to Byte Clocks conversion function, and a Wcdma chips to Byte Clocks conversion function. Event count compare (`EventOffset`) and event modulo terminal count (`EventModulo`) should be less than the event symbol (or frame) period. Event count compare (`EventOffset`) must be less than or equal to the event modulo terminal count (`EventModulo`).

**EvtStrobeSel:** This parameter determines both the RADT counter and start-of-symbol vs. start-of-frame strobe.

**EventModulo:** Event modulo in Byte Clocks, determining the periodicity of this AT2 system event. From IQN2 hardware stand point, after the event has fired once, the event will re-fire every `MOD` clocks. If set to (-1), setting max value which means no additional events between the selected strobcs. Event modulo minimum is mod 16.

**EventMaskLsb:** The system event generator maintains a count which indexes this mask LUT starting with the LSB. This 64-event mask register can be set to mask any combination of 64

events in a symbol. If all 0s, enabling all event time (LLD convention), so LLD sets the mask to 0xFFFFFFFF.

EventMaskMsb: Same as EventMaskLsb.

### 6.5.3 Initialization of AT2 events

The AT2 events are initialized and enabled in two different steps. This allows starting IQN2 hardware first, waiting for system time synchronization, and then starting the SW applications once the RF chain is ready. For this matter, the LLD features 2 APIs, `IQN2_initAt2Event()` and `IQN2_enableAt2Event()`.

Here is an example:

#### User application:

```

egressRadtId = IQN2_getEgressRadioTimerId(iqn2LldObj->aidConfig.firstLteAxC);

at2EventCfg.EventSelect = IQN2_AT2_EVENT_2;
at2EventCfg.EventOffset = offsetForInsertingPackets; // in nanosecs
at2EventCfg.EvtStrobeSel = IQN2_getRadioTimerFrameStrobe(egressRadtId);
at2EventCfg.EventModulo = 500000; // 0.5ms in ns (lte slot)
at2EventCfg.EventMaskLsb = 0xFFFFFFFF;
at2EventCfg.EventMaskMsb = 0xFFFFFFFF;
// Convert to byte clocks prior to IQN2_initAt2Event(), use byte clocks above if 100% accuracy is required
IQN2_initNanoSecsToByteClocks(&iqn2LldObj, &at2Evt2);
IQN2_initAt2Event(&iqn2LldObj, &at2EventCfg);

...

// Configure IQN2 HW
IQN2_startHw(&iqn2LldObj);
// Start BCN timer free running
IQN2_runBcnTimer(&iqn2LldObj);

...

IQN2_enableAt2Event(&iqn2LldObj, IQN2_AT2_EVENT_2);

```

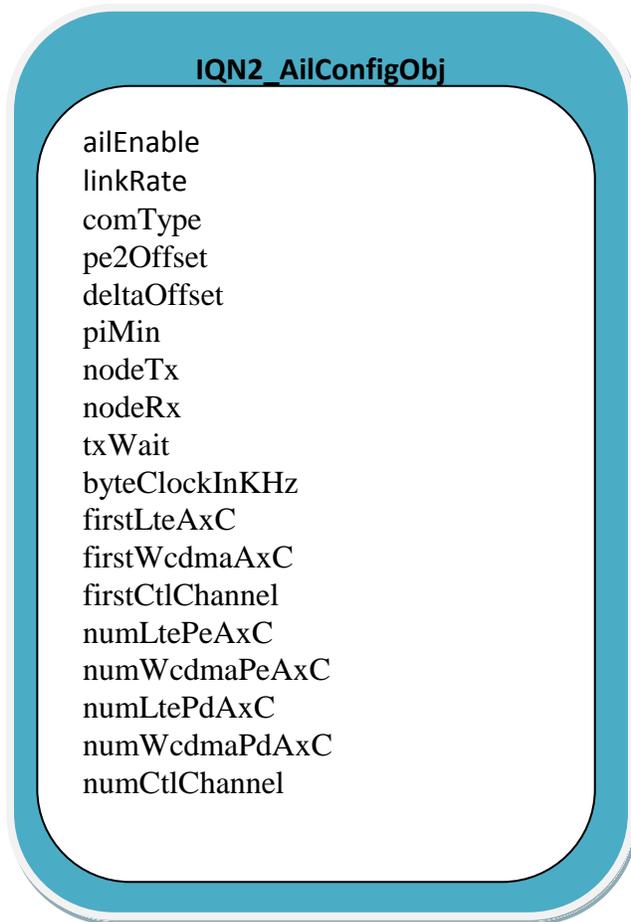
## 6.6 CPRI/OBSAI AIL

### 6.6.1 Definition

The IQN2 AIL (Antenna Interface Link) interfaces to a Radio module via either of CPRI or OBSAI protocols. One AIL module equals one high-speed serialized links (either OBSAI or CPRI). These links can be set at different link rates depending on the application needs. IQN2 AIL can also support combinations of both WCDMA (DL/UL) and LTE (20, 15, 10, 5) modes to interface with RAC/TAC or FFTC.

The IQN2 LLD instantiates `IQN2_AilConfigObj` structure object for each AIL. This structure holds information required to create the CPRI or OBSAI frame to match the radio module configuration.

### 6.6.2 Properties



ailEnable: used to indicate if the ail module needs to be configured or not.

linkRate: holds the link rate of the current AIL module. The link rates tested with the LLD are 4x and 8x. AIL hardware can support {2x, 4x, 5x, 8x, 16x}.

comType: holds the communication type. It can be set to be either SerDes Loopback or communication with an external device. This parameters is used to compute timings in IQN2\_calcParam.c

pe2Offset: pe2Offset is calculated by the function IQN2\_calcParameters(), using the txWait value. This field is used to adjust the timing of the link for the DL side. If not set it will take the default value. See 6.6.5 link parameters.

deltaOffset: deltaOffset is calculated by the function IQN2\_calcParameters(), using the pe2Offset value and the nodeTx value. This field is used to adjust the timing of the link for the DL side. If not set it will take the default value. See 6.6.5 link parameters.

piMin: piMin is calculated by the function IQN2\_calcParameters(), using the deltaOffset value and the nodeRx value. This field is used to adjust the timing of the link for the UL side. If not set it will take the default value. See 6.6.5 link parameters.

nodeTx: NodeTx parameter allows inserting a certain delay based on the place of the KeyStone SoC in the antenna daisy chain. For a direct communication between two devices, the nodeTx value needs to be set to '0'. For the 1st retransmission node, set it to 1, and so on.

nodeRx: NodeRx parameter allows inserting a certain delay based on the place of the KeyStone SoC in the antenna daisy chain. For a direct communication between two devices, the nodeRx value needs to be set to '0'. For the 1st retransmission node, set it to 1, and so on.

txWait: See 6.6.3.

firstLteAxC: holds the number of the first LTE AxC on this AIL. The IQN2 LLD support dual mode, and this fields is used with the numLtePe/PdAxC to separated WCDMA related AxCs from LTE 20, LTE 10 and LTE 5 AxCs.

firstWcdmAxC: holds the number of the first WCDMA AxC on this AIL. The IQN2 LLD support dual mode, and this fields is used with the numWcdmAegress/IngressAxC to separated WCDMA related AxCs from LTE 20, LTE 10 and LTE 5 AxCs.

firstCtlChannel: Holds the number of the first control channel enabled.

numLtePeAxC: Set the total number of LTE AxCs used for egress. It can be LTE 20 AxC and LTE 10 or any other LTE mode.

numWcdmaPeAxC: set the total number of WCDMA AxCs used for egress.

numLtePdAxC: Set the total number of LTE AxCs used for ingress. It can be LTE 20 AxC and LTE 10 or any other LTE mode.

numWcdmaPdAxC: set the total number of WCDMA AxCs used for ingress.

numCtlChannel: total number of control channels used.

### 6.6.3 Tx wait delay

The TxWait holds the maximum expected wait time in nanosecs for:

- PktDMA mode: popping a descriptor and DMA of IQ sample from application memory to AIL egress SI
- DIO mode: TAC processing time (4 chip) and DMA time from TAC or application memory to AIL egress SI

This parameter is used by IQN2\_calcAilTimingForTxNode() to compute deltaOffset. It actually corresponds to PE\_STB value expressed in ns. This value will be used to configure all the link timing parameters (piMin, pe2offset, deltaOffset...).

### 6.6.4 Cpri packing mode

CPRI required a special format where the AxC are interleaved. Regarding the LTE bandwidth, the LLD needs to interleave the containers 8 by 8, 6 by 6, 4 by 4, 2 by 2 or 1 by 1.

LTE bandwidths	cpriPackMode
All	AIF2_LTE_CPRI_1b1
LTE 20 MHz	AIF2_LTE_CPRI_8b8
LTE 10 MHz	AIF2_LTE_CPRI_4b4
LTE 5 MHz	AIF2_LTE_CPRI_2b2

As examples:

- LTE FDD 20 MHz test case, IQN2\_LTE\_CPRI\_8b8 format looks like: {Control, AxC0, AxC0, AxC0, AxC0, AxC0, AxC0, AxC0, AxC0, AxC0, AxC1, AxC1, AxC1, AxC1, AxC1, AxC1, AxC1, AxC1}
- LTE FDD 10 MHz test case, IQN2\_LTE\_CPRI\_4b4 format lookslike: {Control, AxC0, AxC0, AxC0, AxC0, AxC1, AxC1, AxC1, AxC1}

Let's illustrate with a dual mode example. In this example we want 2 LTE 20 AxCs and 16 WCDMA AxCs. The protocol used is CPRI

#### User application:

```
iqn2lldObj.ailConfig[0].ailEnable = 1; //we want to use AIL module 0
iqn2lldObj.ailConfig[0].linkRate = IQN2FL_LINK_RATE_8x; //if we want to have 4AxCs LTE20, we need at least 8x
rate.
iqn2lldObj.ailConfig[0].numWcdmaPeAxC = 16;
iqn2lldObj.ailConfig[0].numWcdmaPdAxC = 16;
iqn2lldObj.ailConfig[0].numLtePeAxC = 2;
iqn2lldObj.ailConfig[0].numLtePdAxC = 2;
iqn2lldObj.ailConfig[0].firstWcdmaAxC = 0; //we configure wcdma AxC first
iqn2lldObj.ailConfig[0].firstLteAxC = 16; //the first LTE AxC follows WCDMA AxCs.
iqn2lldObj.ailConfig[0].pe2Offset = 0; //if not set it will take the default value
iqn2lldObj.ailConfig[0].deltaOffset = 0; //if not set it will take the default value
iqn2lldObj.ailConfig[0].comType = IQN2_COM_SD_LOOPBACK //or IQN2_COM_NO_LOOPBACK
```

### 6.6.5 AIL physical Timing parameters

Within the AIL, there are some internal delays in both egress and ingress directions. The IQN2 LLD allows the user to only specify delayTx and delayRx reference timing parameters and, in turn, configure all other internal delays automatically. If those 2 parameters are set to 0, IQN2 LLD will use default parameters that work with most cases, based on pe2Offset base value.

These timings are computed using the IQN2\_calcParameters() function.

Here are the default values for this offset.

AIL parameters	Default values
Pe2Offset	310
deltaOffset	Pe2Offset + 64 * nodeTx (CPRI) Pe2Offset + 80 * nodeTx (OBSAI)

piMin	deltaOffset + 64 * nodeRx (CPRI) deltaOffset + 80 * nodeRx (OBSAI)
-------	---

### 6.6.5.1 Pi value

If IQN2 is connected with an external device such as a radio head or a CPRI relay solution, an adjustment of Pi value for ingress traffic may be needed, as this external equipment should include a new delay. Pi corresponds to the delay from the physical frame boundary to the actual start of the master frame boundary on the ingress side. It may then include the delay of the RF chain in some cases.

To adjust Pi value on a given system, AIL HW features a Pi capture mechanism in its receive mac module (RM) and these captured values can be read from the AT Pi Captured Value Register for the given AIL module.

#### User application:

```
ailPiCaptured = CSL_FEXT(hIqn2->hF1->regs->Ail[ailNum].AIL_UAT_AIL_REGS.AIL_UAT_PI_BCN_CAPTURE_STS,\
                    IQN_AIL_AIL_UAT_PI_BCN_CAPTURE_STS_RD_VAL);
```

### 6.6.6 Data trace

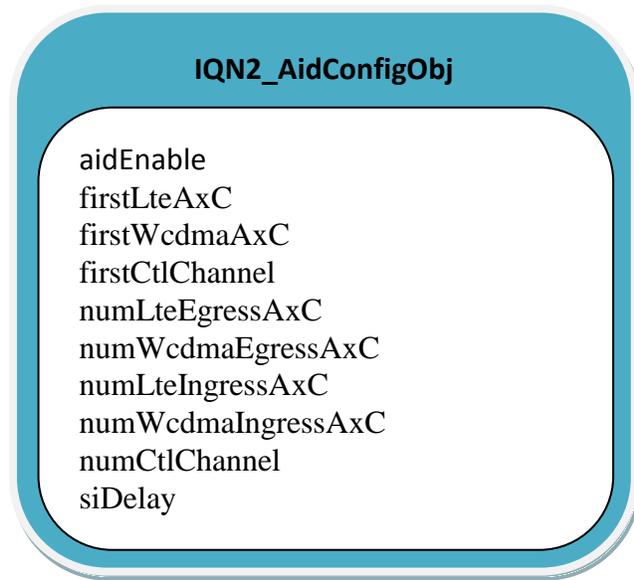
The Data Trace feature allows data from the AIL RM, via an external multiplexer, to be stored in either L2, MSMC, or DDR3 for analysis. This is a debug feature for the AIL only and cannot be used for the DFE/AID. The IQN2 LLD has not implemented that feature yet.

## 6.7 DFE AID2

### 6.7.1 Definition

The IQN2 AID interfaces to a DFE module via the AID/DFE interface. The Ingress block accepts UL data from the DFE and partitions the frames it receives into packets (e.g. symbols for LTE or 8 quad-word packets for DIO). It sends the symbol based traffic to the IQN2 PKTDMA or DIO traffic to IQN2 DIO via a master PKTDMA PSI interface. The Egress block usually reads DL data from system memory or IQN2 DIO via a slave PKTDMA PSI interface and paces the data delivery to the DFE to a 4-sample (chip) strobe from AID uAT RADT.

### 6.7.2 Properties



aidEnable: used to indicate if the aid module needs to be configure or not.

firstLteAxC: holds the number of the first LTE AxC on this AIL. The IQN2 LLD support dual mode, and this fields is used with the numLtePe/PdAxC to separated WCDMA related AxCs from LTE 20, LTE 10 and LTE 5 AxCs.

firstWcdmaAxC: holds the number of the first WCDMA AxC on this AIL. The IQN2 LLD support dual mode, and this fields is used with the numWcdmaEgress/IngressAxC to separated WCDMA related AxCs from LTE 20, LTE 10 and LTE 5 AxCs.

firstCtlChannel: Holds the number of the first control channel enabled.

numLteEgressAxC: Set the total number of LTE AxCs used for egress. It can be LTE 20 AxC and LTE 10 or any other LTE rate.

numWcdmaEgressAxC: set the total number of WCDMA AxCs used for egress.

numLteIngressAxC: Set the total number of LTE AxCs used for ingress. It can be LTE 20 AxC and LTE 10 or any other LTE rate.

numWcdmaIngressAxC: set the total number of WCDMA AxCs used for ingress.

numCtlChannel: total number of control channel used.

siDelay: This value is the equivalent of the TxWait for AID. The siDelay holds the maximum expected wait time in nanosecs for:

- PktDMA mode: popping a descriptor and DMA of IQ sample from application memory to AID egress SI
- DIO mode: TAC processing time (4 chip) and DMA time from TAC or application memory to AID egress SI

Let's illustrate the configuration of the AID structure with a simple LTE 20MHz test case. We want 4 AxCs LTE 20MHz to feed the DFE. Here is one configuration to do that.

### User application:

```
iqn21ldObj.aidConfig.aidEnable = 1;           //we want to use AID module
iqn21ldObj.aidConfig.numWcdmaEgressAxC = 0;
iqn21ldObj.aidConfig.numWcdmaIngressAxC = 0;
iqn21ldObj.aidConfig.numLteEgressAxC = 4;
iqn21ldObj.aidConfig.numLteIngressAxC = 4;
iqn21ldObj.aidConfig.firstWcdmaAxC = 0;
iqn21ldObj.aidConfig.firstLteAxC = 0;        //the first LTE AxC is the first AxC to configure
iqn21ldObj.aidConfig.siDelay = 300;
```

## 6.8 DirectIO engines (DIO2)

### 6.8.1 Definition

The term Direct IO means that a peripheral has dedicated custom logic that implements data movement (as opposed to using EDMA or CPU reads/writes). For IQN2, custom circuitry is built to handle data movement requirements unique to WCDMA.

The DIO2 consists of the following major components:

- Ingress DIO
- Egress DIO
- Data Trace
- SI
  - Ingress SI Interface
  - Egress SI Interface
  - Micro AT module (uAT)
- VBUSM Interface (master)
- Error Event (EE) Module

There are 3 such DIO2 engines in IQN2.

DIO engines are periodic engines tight to dedicated IQN2 timer events as opposed to PktDMA channels which are driven by data arrival. The AT module generates internal system events that control the DIO engine timing. The IQN2 LLD programs these events with the following periodicity:

- Four chips event for TAC/DL
- Eight chips event for RAC/UL

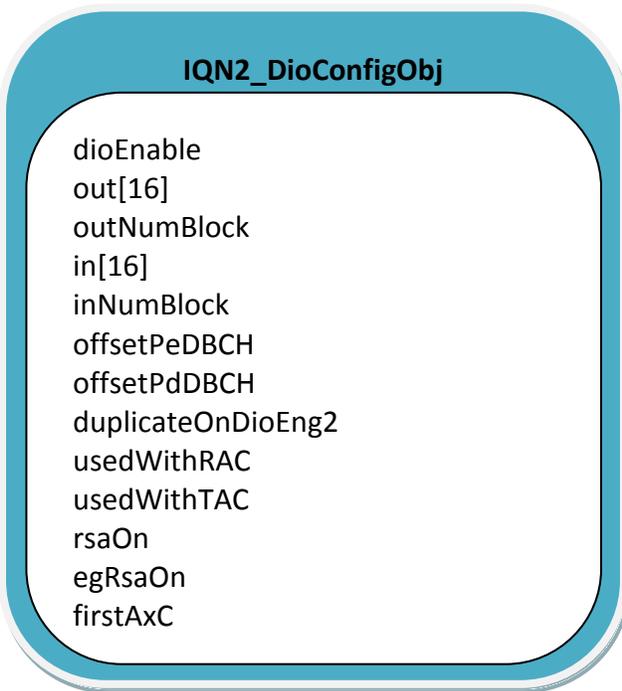
### 6.8.2 Ingress and Egress parameters

Then DIO2 engine programming exposes a certain level of flexibility and the LLD implements it that way.

DIO2 register fields	Description	LLD implementation
dma_num_qwd	Number of QuadWord per AxC	One QW for DL, two QW for UL
dma_num_axc	Number of AxCs associated with the dio engine	Parameter from IQN2_DioConfigObj
dma_vbus_base_addr_axc	Vbus source or destination base address	in/out circular buffer Parameter from IQN2_DioConfigObj
dma_brst_ln	Maximum dma burst length	4 QW
ch_en	DMA channel enable/disable	Set based on number of AxCs for this engine
rsa_cnvrtn_en	Data type selection	1 for UL 0 for DL

dma_num_blks	Number of data blocks to transfer before wrapping back to dma_base_addr	inNumBlock/outNumBlock Parameter from IQN2_DioConfigObj
dma_cfg1_dma_blk_addr_stride	DMA block address stride after transferring each dma block (every event time), the DMA will increment by this amount	Computed by LLD, set to 0x80 if usedWithRAC field in IQN2_DioConfigObj is set to match RAC front end format

### 6.8.3 Properties



dioEnable: There are 3 DIO engines available. Set to one for the DIO to use.

Out[16]: holds buffer start address for each AxCs of this DIO engine in egress direction.

outNumBlock: holds number of DMA blocks (wrap2) for this DIO engine in egress direction.

In[16]: holds buffer start address for each AxCs of this DIO engine in ingress direction.

inNumBlock: holds number of DMA blocks (wrap2) for this DIO engine in ingress direction.

offsetPeDBCH: Holds offset to the first DB channel associated to this DIO engine on transmit side.

offsetPdDBCH: Holds offset to the first DB channel associated to this DIO engine on receive side.

duplicateOnDioEng2: Determines if we use the DIO duplicate feature with the current dio (0= disable, 1= enable). Duplicate feature uses dio engine 2, so this feature cannot be used on both

DIO 0 and DIO 1 at the same time. This duplicate feature can be used for the purpose of debugging DIO traffic going to RAC. (this feature is not yet supported by IQN2 LLD)

usedWithRAC: When in Wcdma mode, tells whether this DIO engine is used with RAC on ingress side.

usedWithTAC: When in Wcdma mode, tells whether this DIO engine is used with TAC on egress side.

rsaOn: Enabled the RSA Data Format Conversion for UL traffic.

egRsaOn: Enabled the RSA Data Format Conversion for DL traffic.

firstAxC: set the first AxC that belongs to that DIO engine. This is used to match the DIO engine with the proper radio timer.(for LTE DIO mode and MIX MODE).

### User application:

```
iqn2LldObj.dioConfig[0].in[num_axc] = ptr_data_buffer_ingress[num_axc];
iqn2LldObj.dioConfig[0].out[num_axc] = ptr_data_buffer_egress[num_axc];
iqn2LldObj.dioConfig[0].inNumBlock = DIO_NUM_BLOCK;
iqn2LldObj.dioConfig[0].outNumBlock = DIO_NUM_BLOCK;
iqn2LldObj.dioConfig[0].usedWithRAC = 0/1;
iqn2LldObj.dioConfig[0].usedWithTAC = 0/1;
iqn2LldObj.dioConfig[0].rsaOn = 0/1; //enable or disable RSA data format conversion in UL
iqn2LldObj.dioConfig[0].egRsaOn = 0/1; //enable or disable RSA data format conversion in DL
```

### 6.8.4 Duplicate feature

With this feature, DIO engine 2 settings are “duplicated” from DIO engine 0 or 1 while another DIO2 circular buffer can be specified. It could be, for instance, to another RAC, or to some DSP memory for debug purposes.

- iqn2LldObj.dioConfig[0/1].duplicateOnDioEng2: 1 means that DIO engine 2 is dedicated to debug mode and will copy this current DIO. IQN2 LLD use DIO engine 2 to duplicate the current DIO. So if DIO duplicate feature is used the application cannot use the DIO engine 2 for traffic purpose. Only one DIO can be debugged at a time, and it can't be DIO engine 2.
- iqn2LldObj.dioConfig[2].in: sets the buffer destination start address for DIO engine 2 .

### 6.8.5 RAC use case

The usedWithRAC parameter in the DIO configuration parameter lets the LLD know that this DIO engine needs to be programmed for the RAC front-end RAM. So it means:

dma\_cfg1\_dma\_blk\_addr\_stride = 0x80; (DIO block stride for each AxC)  
 dma\_num\_blks = 3; ((4 – 1) number of 8-chip blocks, 1 RAC iteration is a 32-chip period)  
 dma\_vbus\_base\_addr\_axc = RAC address for each AxC; (AxCs are interleaved in RAC FEI RAM)

### 6.8.6 TAC use case

The usedWithTAC parameter in the DIO configuration parameter lets the LLD know that this DIO engine needs to be programmed for the TAC RAM. So it means:

dma\_cfg1\_dma\_blk\_addr\_stride = num\_quad\_word\_per\_axc \* num\_axcs;  
dma\_num\_blks = configurable with DIO outNumBlock; (up to (64 – 1) blocks)  
dma\_vbus\_base\_addr\_axc = TAC address for each AxC; (AxCs are interleaved in TAC RAM)

## 6.8.7 DIO reconfiguration using IQN2 Functional Layer

In cases where:

- DIO parameters need to be adjusted by, for instance Physical Layer software
- IQN2 is configured via LLD from ARM Linux or any other software entity external to, for instance Physical layer software

IQN2 functional layer has dedicated hardware control commands and auxiliary APIs that can be used prior to enabling the AT2 events:

```
Iqn2Fl_setupDio2ReconfigureEngine (hIqn2, &dio2ReconfigureEngine); // both Ingr/Egr
Iqn2Fl_setupDio2ReconfigureEgressEngine (hIqn2, &dio2ReconfigureEgrEngine);
Iqn2Fl_setupDio2ReconfigureIngressEngine (hIqn2, &dio2ReconfigureIngrEngine);
```

So let's illustrate with an example:

### Reconfiguration using IQN2 FL example:

```
memset(&dio2EngineSetup, 0x00, sizeof(Iqn2Fl_Dio2ReconfigureEngineSetup));

dio2EngineSetup.engine_idx = 0;

for (idx=0;idx<numWcdmaEgressAxC;idx++)
{
    // address expressed at quadword granularity
    dio2EngineSetup.egress_axc_buffer_start_addr[idx]=(uint32_t)UTILS_local2GlobalAddr(&(wcdma_dio_data[idx][0]))>>4;
}
for (idx=0;idx<numWcdmaIngressAxC;idx++)
{
    // address expressed at quadword granularity
    dio2EngineSetup.ingress_axc_buffer_start_addr[idx]=(uint32_t)UTILS_local2GlobalAddr(&(wcdma_dio_result[idx][0]))>>4;
}

// egress_num_block and ingress_num_block expressed in num_blocks - 1
dio2EngineSetup.egress_num_block      = DIO_NUM_BLOCK - 1;
dio2EngineSetup.ingress_num_block     = DIO_NUM_BLOCK/2 - 1;
// address strides expressed in quadwords
dio2EngineSetup.egress_block_addr_stride = IQN2FL_1QW + 1;
dio2EngineSetup.ingress_block_addr_stride = IQN2FL_2QW + 1;
Iqn2Fl_setupDio2ReconfigureEngine(iqn2LldObj.hFl, &dio2EngineSetup);
```

## 6.9 Control channels

### 6.9.1 Definition

CPRI control words and fast C&M are yet to be defined at LLD level.

DFE CTL is supported, and those channels will be enabled once

iqn2LldObj.aidConfig.numCtlChannel and iqn2LldObj.aidConfig.firstCtlChannel are set.

### 6.9.2 Properties

CPRI control words and fast C&M are yet to be defined at LLD level.

No specific properties for DFE CTL have been identified at this stage.

## 6.10 Synchronization process

### 6.10.1 Mechanism

The AT2 module in IQN2 hardware can detect synchronization from an external source. The possible synchronization sources include:

- RP1 FCB packet (Only valid with using OBSAI RP1 Sync Interface)
- Rad sync input pin (Only valid when using external sync pin)
- Phy sync input pin (Only valid when using external sync pin)
- Pa\_tscomp sync input from NETCP PA (Valid when using IEEE1588 sync)

The IQN2 hardware captures the BCN value when any of the synchronization events occur. Application software may then adjust the BCN offset as needed to achieve RAD timer synchronization.

The difference in sync event selection between AIF2 and IQN2 ATs is that the AIF2 PHY timer is replaced by the IQN2 BCN timer which is always started by software. So SW resynchronization procedure is required and supported by the LLD.

Relying on the IQN2 exception events, an interrupt can be generated upon a sync event occurrence and it is the responsibility of the application software to read the captured offset BCN value and decide whether the IQN2 timer resynchronization procedure from the LLD needs to run. Based on the timerSyncSource of the LLD object instance and the IQN2 AT2 exception flags, the LLD will check that resynchronization is required and apply the new time delay computed by the user application based on the captured offset and system time parameters.

### 6.10.2 SW support

The low-level software resynchronization procedure is described in chapter 7.3.3.2 of the IQN2 hardware user's guide (SPRUHO6). The LLD implements these steps as part of IQN2\_resyncProcedure(). It should be noted, that during this process, all AID2 channels are disabled, as well as all DIO channels and engines. Also this procedure needs to wait for several radio frames in between some of the IQN2 timers adjustments, hence the requirement for the user application to implement the Osal\_iqn2Sleep() callback with 1 to 2ms sleep time. The overall resynchronization procedure is expected to take around six 10-ms radio frames. Another important note is that all radio timers are disabled by this API call, and that the re-enabling of the radio timers needs to be done by the customer application using the IQN2\_enableRadioTimers() API.

Here is an example of one-time resynchronization prior to user application start (AT2 events disabled):

#### User application:

```
Main process:  
iqn2LldObj.timerSyncSource = IQN2_PHY_SYNC;  
...
```

```
// Configure IQN2 LLD instance
IQN2_initHw(&iqn2LldObj, &iqn2InitCfg);
// Configure IQN2 HW
IQN2_startHw(&iqn2LldObj);
// Start BCN timer free running
IQN2_runBcnTimer(&iqn2LldObj);
// Enable sync process if not SW diag sync
if (EXTERNAL_SYNC) {
    // Enable AT2 exception to catch the external sync
    IQN2_enableAt2Exception(&iqn2LldObj, 0);
    // Pend on flag set from the IQN2 exception handler
    while (ext_sync == 0)
    {
        asm (" NOP 5 ");
        asm (" NOP 5 ");
    }
}
...
IQN2_enableAt2Event(&iqn2LldObj, IQN2_AT2_EVENT_2);

Interrupt or periodic process:

/* Do the required servicing here */
IQN2_getException(&iqn2LldObj);

if (iqn2LldObj.timerSyncSource != IQN2_DIAG_SW_SYNC)
{
    // Check if resync is required - doing it once when first sync signal comes
    if (ext_sync == 0) {
        ext_sync = IQN2_resyncProcedure(&iqn2LldObj, 0); // no extra delay adjustments for LLD unit tests
    } else {
        IQN2_disableAt2Exception(&iqn2LldObj, 0);
    }
}
}
```

## 6.11 Runtime configuration

### 6.11.1 AIL Enable/Disable

TBD

### 6.11.2 AxC Enable/Disable

LLD defines APIs to enable/disable individual egress or ingress AxCs, or enable/disable all AxCs at once.

### 6.11.3 AT2 event Enable/Disable

LLD defines APIs to enable/disable individual AT2 events.

## 6.12 Exception counters

The LLD object contains exception flag structures (iqn2EeCount) for all IQN2 exception events. Some of these events are considered as errors and some of these are considered as informative. When the exception comes from an error condition, the eeFlag in iqn2EeCount is set, allowing a simple way to check for error conditions.

The user application will first need to enable the IQN2 exception handling. This can be done in two different ways:

1. A top level function, IQN2\_enableException() can be used to enable all IQN2 events at once. This function will enable all exceptions to be flagged as EV0 and routed to COREPAC\_IQNET\_INT0 interrupt event, with the exception of PKTDMA errors, which

trigger the CIC0\_IQNET\_PKTDMA\_STARVE. So, to handle exceptions, two interrupt events need to be enabled at CPU level.

2. The LLD has a separate exception enable function for all IQN2 modules, for instance `IQN2_enableTopException(hIqn2, interruptEventNum)`, which can be called by the user application. This allows to only enable a subset of the IQN2 exceptions, and also to choose which interrupt event to use for each IQN2 module. `interruptEventNum` could be set to 0 (EV0) or 1 (EV1).

To disable exceptions, the user application can disable the related CPU interrupts, or, for the AT2 event exceptions, use `IQN2_disableAt2Exception()`. This function can be used to support the synchronization procedure, allowing the application to listen to external sync events only during certain period of time.

To gather the IQN2 exception events, a similar approach to the enabling of exceptions is taken. A top level function exists, `IQN2_getException()`, which can be called from a CPU interrupt, or called periodically. This function first checks the exception origin registers and flags, to only call the module exception functions which caused interrupt events to be raised. Individual module exception functions can also be called by the user application. The exception counters are incremented by these functions.

A snapshot of the current exception counters can be captured by calling, `IQN2_captureException()`. Exception counters can be reset by calling `IQN2_resetException()`.